APPENDIX

This appendix contains listings for functions and data structures in C programming language, for use in a rover in an embodiment of the invention such as described above. The functions typical would be periodically executed in a rover to check any alert conditions of entries in a proximity queue and update the proximity queue to remove expired entries.

```
/*********************************************************/
/* prx_alrt.c                                            */
/* Function calls for proximity alert monitoring in the  */
/* rover.                                                */
/* S Taylor, @Road Inc.                                  */
/*********************************************************/

#include"rover_ds.h"

#define D_HYSTERESIS (float) 0.2  // 10% either side hysteresis
                                  // for points tripping in either
                                  // direction
#define TRIP_HYSTERESIS_TIME 120  // seconds


/*********************************************************/
// Function to check for tripping of entries in the ordered
// proximity queue, and deleting any expired points.  This
// function would typically be called every 10 seconds
/*********************************************************/
void check_ordered_proximity(long current_lat, long current_lon,
             ULONG current_time){

t_proximity_Q_entry *p_entry;
short need_resort;

    need_resort = 0; // assume we don't need to resort the queue
    p_entry = ordered_proximity_queue; // point at head of queue

    // first check for empty queue
    if(p_entry->tag_id){
       // check the head of the queue to see if it was tripped
       // see listing below for check_point_tripped function
       if(check_point_tripped(p_entry, current_time)){
            need_resort = 1;
       }

        // check the remainder of the queue to see if any points
        // have time expired
        for(i=1;i<SZ_ORDERED_PROX_Q;i++){
            p_entry = &ordered_proximity_queue[i];
            // check for time expiry of point
            if((p_entry->tag_id) &&
                   (p_entry->expiry_time < current_time)){
               // delete point (set to all zero)
               memset(p_entry, 0x00, sizeof(t_proximity_Q_entry));
               need_resort = 1;
            }
        }

        if(need_resort){  // resort the queue
          // sort by E.T.A. (smallest ETA at top of list),
```

```
                // placing empty entries at end of queue.
                sort_by_ETA(ordered_proximity_queue);
            }
        }
}

/*********************************************************/
// Function to check for tripping of entries in the unordered
// proximity queue, and deleting any expired points.  This
// function would typically be called every 30 seconds
/*********************************************************/
void check_unordered_proximity(long current_lat, long_current_lon,
            ULONG current_time){

t_proximity_Q_entry *p_entry;

    // go through all points in queue checking for trip
    for(i=0;i<SZ_UNORDERED_PROX_Q;i++){
        p_entry = &unordered_proximity_queue[i];     // point at entry
        // check for empty entry
        if(p_entry->tag_id){
            // see listing below for check_point_tripped function
            check_point_tripped(p_entry, current_time);  // check entry
        }
    }
}

/*********************************************************/
// Function to check data in a point, and add to the
// appropriate queue, resorting if necessary.
// Returns 0 if successful, 1 if point bad, -1 if queue full
/*********************************************************/
short add_new_point(t_proximity_Q_entry *pnew_entry,
            ULONG current_time){

short return_val;
t_proximity_Q_entry *p_entry;

    return_val = 0;

    // check that expiry time has not already been met,
    // that non-zero tag_id was assigned and that
    // will trip on at least one day
    if( (pnew_entry->expiry_time < current_time) ||
        (pnew_entry->tag_id == 0) || (pnew_entry->days_to_trip == 0) ){
       printf("Attempt to add expired/null tag entry to prox Q\n");
       return(1);  // point data was bad
    }

    // Also need to check for duplicate tag_id :
    // especially if resynchronizing with server
    // If duplicate is found then delete it.
    for(i=0;i<SZ_UNORDERED_PROX_Q;i++){
       p_entry = &(unordered_proximity_queue[i]);
       // identical tag_id indicates duplicate point
       if( (p_entry->tag_id == pnew_entry->tag_id) ){
           // delete existing entry
           memset(p_entry, 0x00, sizeof(t_proximity_Q_entry));
       }
    }

    for(i=0;i<SZ_ORDERED_PROX_Q;i++){
       p_entry = &(ordered_proximity_queue[i]);
       // identical tag_id indicates duplicate point
       if( (p_entry->tag_id == pnew_entry->tag_id) ){
           // delete existing entry
           memset(p_entry, 0x00, sizeof(t_proximity_Q_entry));
       }
```

```
        }

        // Now determine which queue entry must be added to
        // if trip_when_head is set then add to the ordered Q
        // so that point is only monitored when head of queue.
        // if trip_when_head == 0 then add to unordered queue
        // for constant monitoring.
        if(pnew_entry->trip_when_head){

           // add entry to ordered queue
            for(i=0;i<SZ_ORDERED_PROX_Q;i++){
               p_entry = &(ordered_proximity_queue[i]);
               if(p_entry->tag_id){    // queue entry is in-use
                  continue;
               } else {
                  // add entry to the queue
                  memcpy(p_entry, pnew_entry, sizeof(t_proximity_Q_entry));
                  // resort the queue
                   sort_by_ETA(ordered_proximity_queue);
                  return(0);  // success
               }
            }

        } else {
           // add entry to unordered queue
            for(i=0;i<SZ_UNORDERED_PROX_Q;i++){
               p_entry = &(unordered_proximity_queue[i]);
               if(p_entry->tag_id){    // queue entry is in-use
                  continue;
               } else {
                  // add entry to the queue
                  memcpy(p_entry, pnew_entry, sizeof(t_proximity_Q_entry));
                  return(0);  // success
               }
           }
        }
        return(-1);  // indicate queue was full
    }

    /*********************************************************/
    // Function to find a point in one of the queues and
    // delete it.  Allows server to delete points for
    // whatever reason.  Zero tag id indicates delete all points.
    // Returns 0 if successful, 1 if point not found
    /*********************************************************/
    short delete_existing_point(ULONG match_tag_id){

    short return_val;
    t_proximity_Q_entry *p_entry;

        return_val = 1;  // assume failure
        // this flag allows server to specify deletion of
        // all proximity points from both queues
        if(match_tag_id == 0){
           memset(ordered_proximity_queue, 0x00,
                  SIZE_ORDERED_PROX_Q*sizeof(t_proximity_Q_entry) );
           memset(unordered_proximity_queue, 0x00,
                  SIZE_UNORDERED_PROX_Q*sizeof(t_proximity_Q_entry) );
           return_val = 0;
        }

        // else find point in either queue : if found then delete
        else {
            for(i=0;i<SZ_ORDERED_PROX_Q;i++){
               p_entry = &(ordered_proximity_queue[i]);
               // if queue entry is in-use
               if(p_entry->tag_id == match_tag_id){
                   // delete entry from the queue
```

```
                memset(p_entry, 0x00, sizeof(t_proximity_Q_entry));
                sort_by_ETA(ordered_proximity_queue); // sort by ETA
               return_val = 0;
               break;
            }
         }
         if(return_val){
              for(i=0;i<SZ_UNORDERED_PROX_Q;i++){
                p_entry = &(unordered_proximity_queue[i]);
               // if queue entry is in-use
                if(p_entry->tag_id == match_tag_id){
                    // delete entry from the queue
                    memset(p_entry, 0x00, sizeof(t_proximity_Q_entry));
                  return_val = 0;
                  break;
                }
            }
         }
      }
      return(return_val);
   }


   /*********************************************************/
   // Function to check whether point is tripped/expired.
   // If point is deleted then non-zero is returned, otherwise
   // function returns zero.
   /*********************************************************/
   short check_point_tripped(t_proximity_Q_entry *p_entry,
                ULONG current_time){
   ULONG distance = 0;  // records current distance
   ULONG inner_radius, outer_radius; // used to implement hysteresis
   short return_val;  // 1 if point was deleted
   short point_tripped;  // 1 if point is tripped

       return_val = 0;
       point_tripped = 0;

       // check for time expiry of point
       if(p_entry->expiry_time < current_time){
          // delete point (set to all zero)
          memset(p_entry, 0x00, sizeof(t_proximity_Q_entry));
          return(1);
       }

       // Next check that it is a sufficiently long time since the
       // point was last tripped
       if ((current_time - p_entry->time_last_tripped) <
            TRIP_HYSTERESIS_TIME ){
          // still within hysteresis time : just return
          return(0);
       }

       // If you get to here then need to check for tripping of point

       // calculate current distance from point
       distance = calculate_distance( current_lat, current_lon,
          p_entry->center_lat, p_entry->center_lon);

       // allow a %age hysteresis where delete_when_tripped == 0
       // to prevent multiple rapid alerts for rover moving
       // tangentially to radius.
       if( delete_when_tripped == 0x00 ){
           inner_radius = p_entry->radius * ( 1 - D_HYSTERESIS/2);
           outer_radius = p_entry->radius * ( 1 + D_HYSTERESIS/2);
       } else {
           // use unadultered value if point will be deleted when tripped
           inner_radius = p_entry->radius;
```

```
            outer_radius = p_entry->radius;
        }

        // look for perimeter being traversed by an approaching
        // rover.  The previous cross of the perimeter must have
        // been in the opposite direction (to add hysteresis).
        if( (distance < inner_radius) &&    // current dist < radius
            ( (p_entry->direction_last_crossed == RECEDING) ||
              (p_entry->direction_last_crossed == NOT_YET_CROSSED) ) ){

           // record that the perimeter has been broken in this direction
           // this is used for future hysteresis calculations
           p_entry->direction_last_crossed = APPROACHING;

           // check if entry is activated by traverse in this direction
           if(p_entry->trip_direction & APPROACHING)
               point_tripped = 1;

        } else

        // look for perimeter being traversed by a receding
        // rover.  The previous cross of the perimeter must have
        // been in the opposite direction (to add hysteresis)
        if( (distance > inner_radius) &&    // current dist > radius
            ( (p_entry->direction_last_crossed == APPROACHING) ||
              (p_entry->direction_last_crossed == NOT_YET_CROSSED) ) ){

           // record that the perimeter has been broken in this direction
           // this is used for future hysteresis calculations
           p_entry->direction_last_crossed = RECEDING;

           // check if entry is activated by traverse in this direction
           if(p_entry->trip_direction & RECEDING)
               point_tripped = 1;

        }

        //check for tripping of point
        if(point_tripped){
            tx_prox_alert(p_entry->tag_id);  // transmit alert to server

            // check if point must be deleted
            if(delete_when_tripped){
               memset(p_entry, 0x00, sizeof(t_proximity_Q_entry)); // kill
               return_val = 1;  // so that calling function can resort
            }
        }

        return(return_val);
    }

    /*********************************************************/
    /* rover_ds.h                                            */
    /* Datastructure definitions for the rover for proximity */
    /* alert indications.                                    */
    /* S Taylor, @Road Inc.                                  */
    /*********************************************************/

    typedef unsigned long ULONG;  // 32 bit unsigned integer
    typedef unsigned short WORD;  // 16 bit unsigned integer
    typedef unsigned char BYTE;  // 8 bit unsigned integer

    // Bit patterns for days_to_trip field
    #define BP_SUNDAY    0x01
    #define BP_MONDAY    0x02
    #define BP_TUESDAY   0x04
    #define BP_WEDNESDAY 0x08
    #define BP_THURSDAY 0x10
```

```
#define BP_FRIDAY   0x20
#define BP_SATURDAY 0x40

// Bit patterns for direction variables
#define APPROACHING 0x01
#define RECEDING    0x02
#define NOT_YET_CROSSED 0x00

/************** Proximity queue entry ************************/
// Examples of field combinations for typical applications :
//
// For home / warehouse delivery :
//      center_lat, lon = destination
//      expiry_time = end of today
//      radius = 1-10 miles (in meters)
//      estimated_time_arrival = set by dispatcher reflects delivery order.
//      delete_when_tripped = 1
//      trip_when_head = 1
//      trip_direction = APPROACHING
//      Days to trip = either set to today, or to 0xFF
//      Description = customer name, address, order number
//
// For stolen vehicle alarm : (vehicle in normal use Mon-Fri)
//      center_lat, lon = truck depot
//      expiry_time = 0xFFFFFFFF
//      radius = X miles (in meters)
//      estimated_time_arrival = N/A since trip_when_head = 0
//      delete_when_tripped = 0
//      trip_when_head = 0  always trip
//      trip_direction = 0x02 when receding, or 0x03 either direction
//      Days to trip = 0x41 (binary) 01000001 (Sat and Sun)
//      Description = Stolen vehicle alarm (threshold distance + days)
//
// When downloading points from the server to the rover the data
// packet must give a value for each of these fields.  It is expected
// that the data structure in the server is a superset of the
// information in the data structure presented here.
// The server would store all information below, and would additionally
// store the contact information for the alert, whether, e-mail,
// telephone call, of page etc.  The server could additionally link
// this data to a database of customer orders, the format of which is
// beyond the scope of this file.
//
typedef struct proximity_Q_entry{

    ULONG tag_id;  // uniquely identifies an alert : 0 equals empty Q
entry
    char  description[SZ_PRX_TXT];  // for use of the driver
    long  center_lat;  // latitude of destination in MAS
    long  center_lon;  // longitude of destination in MAS
    ULONG radius;      // radius of trip in meters
    // all times in seconds, relative to 0/0/97
    ULONG estimated_time_arrival;  // used for sorting points in queue
    ULONG expiry_time;             // point is deleted at this time
    BYTE  delete_when_tripped;  // if 1 then point is deleted when tripped
                                // if 0 then point deleted at expiry time
only
    BYTE  trip_when_head;  // if 1 then point is only tripped when top of
queue
                           // if 0 then point is tripped whenever crossed
    BYTE  trip_direction;   // 0x01 : trip when rover is approaching
                            // 0x02 : trip when rover is receding
                            // 0x03 : trip in both directions
    BYTE  days_to_trip;     // bit field for 7 days of week, Sun-Sat

    // These final fields are not be downloaded from the server.
    // They are used locally in the rover to generate hysteresis
    // to prevent multiple trips on a point in rapid succession.
```

```
    // Hysteresis values may or may not be sent from the server
    // on a per-entry basis.
    ULONG time_last_tripped;  // time hysteresis
    BYTE  direction_last_crossed;  // direction in which rover was moving
                                   // when perimeter was last crossed
                                   // 0x01 : approaching
                                   // 0x02 : receding
                                   // 0x00 : has never been tripped (init)
} t_proximity_Q_entry;

// Extern declarations for two queues :
// one for points with trip_when_head == 1 (i.e. only check top entry)
// the second for points with trip_when_head == 0 (i.e. check every entry)
extern t_proximity_Q_entry ordered_proximity_queue[SZ_ORDERED_PROX_Q];
extern t_proximity_Q_entry unordered_proximity_queue[SZ_UNORDERED_PROX_Q];
```